AUTOMATED VULNERABILITY DISCOVERY IN BOTNET COMMAND AND CONTROL INFRASTRUCTURE

A Thesis Presented to The Academic Faculty
By Ehsan Muzaffar Asdar

In Partial Fulfillment of the Requirements
for a Bachelor of Science in Computer Science
in the College of Computing

Georgia Institute of Technology
April 2020

Introduction

Botnets have become a common and costly adversary for large corporations and governments. Hosts infected with botnet malware communicate to a central command and control server, from which bot developers can launch coordinated malicious attacks. Security organizations across the globe employ researchers to triage and takedown botnet systems through careful reverse engineering and analysis. These takedowns are costly due their reliance on manual analysis, and do not scale to the enormous number of botnet variants that are circulating and regularly discovered [4]. Researchers are fighting an uphill battle to contain the damage by botnets to computer systems around the world, and are in need of more effective, automated approaches to aid in rapidly executing botnet takedowns.

Automated vulnerability discovery is nothing new to the world of computer security. There is a wealth of research on methods of identifying vulnerabilities in a range of computer software [5,6]. Yet, in the research community, these techniques have seldom been applied to discover vulnerabilities in malware. In our research, we explore how recent advancements in binary analysis systems can be used to surface vulnerabilities in malware. Specifically, we look at how automated understanding of botnet clients can be used to uncover vulnerabilities in a command and control server.

Over the course of our research, we have looked at many different methods for uncovering vulnerabilities in client software. In this thesis, I focus on the development of one such method which looks to fingerprint versions of free and open source software (FOSS) libraries. These widely available libraries are commonly used by malware authors during development. I explore cases where these FOSS libraries used by a botnet client must necessarily constrain the version of the same FOSS library running on a command and control server (i.e. the outdated client does not support connecting to newer versions of the server software). This, in cases we analyze, can open the command and control server to possible attacks, and this form of analysis readily lends itself to automation. Our overall work in this area remains preliminary - the vulnerability detection technique I discuss is one of several that will be included in our final research.

Background

In order to receive instructions, botnet clients must communicate over a network to a command and control server. During preliminary exploration of our malware dataset, we noted that many samples used open source client libraries to facilitate this communication. This is not surprising, as free and open source software (FOSS) libraries have been shown in studies to be commonly used by malware authors - so much so that they are often used as a method for categorizing and chronicling the lineage of malware families [1].

FOSS libraries, as all complex software, are subject to periodic vulnerability disclosures. Several tools are available to automatically check systems for out of date libraries and recommend remediation steps. Cve-bin-tool, a binary analysis tool developed by Intel, is one such tool. When run on a binary, it fingerprints versions of common FOSS libraries used by a given executable, and presents known vulnerabilities affecting the detected version [9].

In any client-server system, software engineers must work hard to ensure compatibility. Since clients and servers can be updated independently, authors typically standardize on a *protocol*, which defines the proper interaction between the two entities. As long as a protocol is always followed, the client and server can be updated independently and still remain compatible. Inevitably, however, a protocol will need to be modified which breaks compatibility with old versions of the client and server software. Breaking compatibility can have a range of consequences. Oftentimes, an incompatible client and server will refuse to establish connection. In other instances, the effects may be more subtle, such as a degraded set of supported features or reduced speed of communication.

Compatibility issues lead to the creation of bands of software versions in many popular network libraries, where certain combinations of servers and clients are compatible (when they support the same protocol version) or incompatible (when a client supports different protocol versions from the server). Since botnet clients must be able to communicate with a command and control server for proper operation, the two must support the same version of the protocol they use to communicate. So, by detecting the version of a protocol used by a client we can *constrain* the possible range of protocol versions that the server is utilizing. It's important to note that the version of a FOSS library may not necessarily match the protocol version; many updates will often be released for a library without the underlying protocol version changing.

We divide these constraints into two types:

- A *partial constraint* is a range of server versions that a client will permit connections to but with reduced functionality or missing features.
- A *full constraint* is a range of server versions that a client will refuse to connect to. It would be *impossible* for a command and control server to be running one of these versions, as the botnet client would not be able to connect or properly interpret commands.

Constraints on server versions can be both *forward* (for releases created after the release date of the client software) and *backward* (for releases created prior to the release date of the client software). An example of possible constraints is shown in the table below. Version numbers follow the semantic versioning scheme, where a greater major, minor, or patch number indicates a more recent release.

Constraint examples for software library libfoo

Client Version	Server Version	Type	Comment
2.X.X	3.X.X+	Forward Full Constraint	Breaking change introduced in version 3.0, 2.X.X clients cannot connect
2.X.X	1.X.X	Backwards Partial Constraint	2.X.X client supports features not supported by 1.X.X servers
1.X.X	2.X.X-3.X.X	Forward Partial Constraint	2.0 server introduces new features not supported by older clients
1.X.X	3.X.X+	Forward Full Constraint	Breaking change introduced in 3.0, 1.X.X clients cannot connect

Since FOSS library authors often operate as volunteers with limited resources, it is common for FOSS libraries to define a support period for a library version. After this support period, the unsupported version (called "end of life") is typically no longer updated with security patches and thus accumulates an attack surface as new vulnerabilities in the library are discovered. For our research, we look to detect end of life versions of FOSS libraries in our client that have a *forward full or partial constraint* on all currently supported versions of the software. For partial constraints, we look at cases where the unsupported features are security-related and increase attack surface. For full constraints, samples matching these properties indicate that the corresponding server component must also be on an end of life version and thus possibly vulnerable to exploits of unpatched vulnerabilities.

In the rest of this paper, we discuss several examples of constraints in popular FOSS libraries. Then, we develop a system to automatically detect samples utilizing vulnerable library versions in our malware dataset and evaluate the performance of our automated vulnerability discovery technique.

Libraries Tested

For the purposes of this thesis, we chose several candidate libraries to research from a large list of open source software libraries [10]. For each library, we public documentation provided by FOSS library authors to determine which library versions introduced a breaking protocol change. From this documentation, we developed a set of candidate constraints. The constraints were then confirmed through analysis of library source code and functional testing of relevant library versions detailed below. We briefly discuss the libraries investigated broken down by constraint type.

Partial Constraints

Software	Client Version	Server Version	Type
Libssl	0.9.8 and prior	All	Partial Forwards and Backwards Constraint
Libopenvpn	2.2.0 and prior	All	Partial Forwards and Backwards Constraint

As examples of partial constraints we focused on end of life versions of two popular network cryptography libraries. Old cryptography client libraries are often forward compatible with newer web servers, but only support cipher suites for authentication and authorization that are deemed outdated and unsafe by the security community.

Libssl

Libssl, developed by the OpenSSL project, is a popular library for securing network communications between a client and server. Libssl 0.9.8 and prior have a maximum supported TLS version of TLS 1.0, which is widely considered end of life and susceptible to denial-of-service and information disclosure attacks [7,8,11]. Consequently, any server accepting connections from a 0.9.8 client must, at best, communicate with the client using TLS 1.0. Thus, there is a high probability of success in denial of service or information disclosure attacks against such a server.

Libopenvpn

OpenVPN is a popular library for creating virtual private network (VPN) connections. OpenVPN 2.2.0 and prior are end-of-life, and OpenVPN versions prior to 2.3.3 have a maximum supported TLS version of TLS 1.0 [12,13]. As discussed above, this leads botnets that rely on VPN servers vulnerable to information disclosure or denial of service attacks.

Full Constraints

Software	Client Version	Server Version	Type
Libmongoc	1.0 or prior	3.X.X+	Full Forward Constraint
Libmysqlclient	4.1.0 or prior	5.6+	Full Forward Constraint

As examples of full constraints we look at two examples of popular database libraries that have undergone breaking protocol changes at some point in their history. These protocol changes force a partition - libraries with a version at/below the version of protocol change are incompatible with newer server versions.

Libmongoc

MongoDB underwent access control changes in the database's 3.0 release. Thus, shells and client libraries on versions below 3.0 are not compatible with 3.0 when access control is enforced, and all clients below 3.0 are now end of life [14,15].

Libmysqlclient

MySQL 4.1.0 and older use an outdated authentication mechanism that is not supported in clients 5.6 and newer [16]. MySQL 5.6 is end of life [17]. Clients running 4.1.0 and below cannot connect to servers 5.6 and more recent.

Design

Our goal for this research is to introduce methods of automatically identifying vulnerabilities in botnet command and control infrastructure. The constraints mentioned above provide a framework for identifying server vulnerabilities with only the knowledge of a FOSS library version utilized by a client. In this section, we develop a system capable of rapidly analyzing malware samples, determining the client version of any FOSS libraries they use, and matching these clients to constraints.

Static Linking

In a statically linked binary, all relevant library routines used by the binary are included alongside the binary code in a single executable package. The advantage of this approach is portability - the author does not have to rely on any system-installed packages for the binary to function properly. For this reason, statically linking libraries is common by malware authors. In

our work here, we limit ourselves to developing a robust method of fingerprinting library versions in statically linked binaries. Our work to fingerprint dynamically linked libraries included by a malware author is ongoing.

Detector Development

For each FOSS library we are testing, we wrote regular expressions to match known strings contained in binaries using the library and associated these with the proper library version. We developed two detectors for each library, one to match any version of the library (a "library detector"), and one to specifically match versions satisfying our constraint (a "constraint detector"). These two detectors work in tandem - once a library is detected, the constraint detector is used to validate if the library version is vulnerable

Library detectors for libraries tested

Library	Rule
Libssl	(?i)part of openssl[a-z]*\d\.\d\.\d
Libopenvpn	(?i)openvpn[a-z]*\d\.\d\.\d
Libmongoc	(?i)mongoc
Libmysqlclient	(?i)mysql[a-z]*\d\.\d\.\d

Constraint detectors for libraries tested

Library	Rule
Libssl	(?i)part of OpenSSL[a-z]*0\.[0-8]\.[0-9]+[a-z]*, (?i)part of OpenSSL[a-z]*0\.9\.[0-8][a-z]*, (?i)OpenSSL[a-z]*0\.[0-8]\.[0-9]+[a-z]*, (?i)OpenSSL[a-z]*0\.9\.[0-8]+[a-z]*
Libopenvpn	(?i)OpenVPN[a-z]*[01]\.[0-9]+, (?i)OpenVPN[a-z]*2\.[01]
Libmongoc	(?i)db version[a-z]*v?[0-2]\.[0-9]+
Libmysqlclient	(?i)mysql[a-z]*[0-3]\.[0-9]+, (?i)mysql[a-z]*4\.[0-1]\.0

Results

Dataset

Our dataset, received from industry and academic collaborators, contains 190,319 Windows Portable Executable format malware samples. Our experiments were run on the entire dataset.

Experimental Setup

Our experiments were run on a virtual machine in a lab-owned cluster with 16 virtual cores and 48 GB of RAM. Each binary was checked by the execution engine for a match with the detectors mentioned, and overall results were tallied. Then, 100 samples matching each library detector and constraint detector were chosen at random. These matches were checked for accuracy by manually disassembling and inspecting the sample in question. These results were used to assess the accuracy of the detector. For detectors matching less than 100 samples, all matches were inspected.

Discussion

Table 1 - Library Detector Effectiveness

Detector	**Detections**	**TP**	**FP**
Libssl	100	91	9
Libopenvpn	42	37	5
Libmongoc	1	1	0
Libmysqlclient	100	98	2

Table 2 - Constraint Detector Effectiveness

Detector	**Detections**	**TP**	**FP**
Libssl	100	98	2
Libopenvpn	0	N/A	N/A
Libmongoc	0	N/A	N/A
Libmysqlclient	67	67	0

Detector Effectiveness

Our results found the Libssl library detector to detect 91 true positives out of 100 samples, the Libopenvpn detector to detect 37 true positives out of 42 samples, the Libmongoc detector to detect 1 true positive out of 1 sample and the Libmysqlclient to detect 98 true positives out of 100 samples. The constraint detector for Libssl had 98 true positives and 2 false positives, and the Libmysqlclient constraint detector was completely accurate. The constraint detectors for Libopenvpn/Libmongoc detectors were unable to be evaluated as they both did not flag any samples.

While the library detectors were moderately accurate, there were several matches that the detectors miss-classified. In the cases we analyzed, this was due to stray strings included in the binary that happened to be the same as strings that were tested by our detectors. This is certainly a weakness of this detector approach - a more robust detection system would confirm usage of the library by analyzing a control flow graph of the target sample to ensure library functions are being called. We are developing a system to incorporate this analysis now, and we hope to use such techniques for detectors in our final research.

Constraint detectors were found to have a higher degree of accuracy than library detectors. This is likely due to the tighter restrictions on the regex to only match certain specific version ranges and strings in the samples analyzed. That said, these detectors are still susceptible to the same issue as the library detectors. In our framework, a valid library detection is required before a constraint detector is run (the library detector must flag positive). Thus, as the above-mentioned improvements are implemented, the constraint detector should improve, as it will receive higher quality input from the improved library detectors.

Our overall goal for this work is to automatically surface previously unseen vulnerabilities that could be used to disrupt botnet activity. Each true positive constraint detected represents an automatically discovered botnet vulnerability. However, some samples vulnerable to our techniques may go undetected (i.e. *false negative* results). Due to the low rate of detection over a large dataset, it was infeasible to determine a precise false negative rate, but we were able to identify probable classes of false negatives based on specific samples observed in our dataset:

- Other libraries implementing the same protocol, e.g. other client libraries for MySQL other than the most popular Libmysqlclient. Our approach tags samples based on specific signatures in the library under test and did not catch these samples even though they satisfy the same constraints. In future work, we plan to solve this by expanding the set of detectors to encompass multiple popular libraries for each protocol.
- Variants of libraries under test. OpenSSL, in particular, is included in binaries in a wide variety of formats. While it is frequently used by the top-level binary code, it may also be included as a sub-dependency of another library. For the vast majority of cases, we observed that our detectors matched these variants properly. However, there were cases

observed where a binary used Libssl core functions but no longer included the strings matched by our detector. In future work, we plan to solve this by incorporating more sophisticated techniques for detector fingerprinting, including hashing of library assembly instructions.

Table 3 - Detected Constraints

Library	**Constraint**	**Library Detections**	**Constraint Detections**	**Percentage of samples with library that were vulnerable**	**Percentage of dataset that was vulnerable**
Libssl	0.9.8 and prior, Partial Forwards and Backwards Constraint	4838	466	9.6%	0.20%
Libopenvpn	2.2.0 and prior, Partial Forwards and Backwards Constraint	42	0	0%	0%
Libmongoc	2.X.X and earlier, Full Forward Constraint	1	0	0%	0%
Libmysqlclient	4.1.0 and prior, Full Forward Constraint	600	67	11.67%	0.04%

Detections for each library constraint, after being run on our dataset, are shown above. The table also shows the number of detections as a percentage of the total dataset (190,319 samples), samples that contained the FOSS library at any version, and the constraint detections as a percentage of library detections. We detected 466 samples including Libssl and 67 samples including Libmysqlclient that are vulnerable to our constraints. Each detection represents a unique, previously unknown vulnerability in a botnet that could be used in more quickly understanding and taking down botnet infrastructure. While we detected 0 instances of Libopenvpn and Libmongoc constraints, we believe this likely due to the small percentage of

Libopenvpn and Libmongoc samples in our total dataset. Work is ongoing to grow the size of our dataset under review and add additional libraries to this framework.

Future Work

We believe we are early in our research into this subject, and there are several areas of future work.

Benign Software

While the techniques discussed are applied to malware, the same techniques could also be used to alert software developers of possible vulnerabilities in their software. We believe that the sort of software scanning detailed in this paper could be a very powerful tool to proactively identify and prevent security incidents before they occur.

Malware Families

Studies of modern botnets have shown that they are often decentralized - it is possible for many variants and versions of malicious bots to interoperate with each other [2,3]. Some of these bots may use libraries or library versions that are more vulnerable than others (for example, an older release). Thus, if it is possible to effectively establish the lineage of a malware sample, it may also be possible to create a superset of possible FOSS vulnerabilities in the server based on the most severe vulnerabilities present in all variants of the botnet client.

Library Constraints

For the experiments conducted, we consider the version constraints discussed in the strictest possible sense - and limit our exploration to cases where we can be completely assured that a vulnerability exists. However, the library versions also tell us something interesting about the binary - a ballpark estimation of when it was created or last patched. The time at which the client was created may aid in studies of botnet lineage, and also allow researchers to target possible attacks against a server with a higher probability of success, even if such attacks are not assured to succeed.

Other Vulnerability Classes

This exploration of FOSS library vulnerabilities is just one of several types of rules that will be released in our final work. All vulnerability systems fall under the same overall umbrella - using knowledge of a botnet client to disrupt the behavior of command and control infrastructure.

Acknowledgements

Thank you to Professor Brendan Saltaformaggio, Professor Alessandro Orso, and PhD Student Jonathan Fuller for advising on this project.

References

[1] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. 2018. FOSSIL: A Resilient and Efficient System for Identifying FOSS Functions in Malware Binaries. Retrieved April 12, 2020 from https://doi.org/10.1145/3175492

[2] Dennis Andriesse, Christian Rossow, Brett Stone-Gross, Daniel Plohmann, and Herbert Bos. 2013. Highly resilient peer-to-peer botnets are here: An analysis of gameover zeus. In *2013 8th International Conference on Malicious and Unwanted Software:" The Americas"(MALWARE)*, IEEE, 116–123.

[3] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, and Michalis Kallitsis. 2017. Understanding the mirai botnet. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 1093–1110.

[4] Ron Condon. Symantec Internet threat report highlights botnet, malware trends. *ComputerWeekly.com*. Retrieved April 12, 2020 from https://www.computerweekly.com/news/1510769/Symantec-Internet-threat-report-highlights-botnet-malware-trends

[5] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. 2017. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Comput. Surv. CSUR* 50, 4 (2017), 1–36.

[6] Bingchang Liu, Liang Shi, Zhuhua Cai, and Min Li. 2012. Software vulnerability discovery techniques: A survey. In *2012 fourth international conference on multimedia information networking and security*, IEEE, 152–156.

[7] Vincent Lynch. 2018. Deprecating TLS 1.0 & 1.1 | DigiCert Blog. *DigiCert*. Retrieved April 13, 2020 from /blog/depreciating-tls-1-0-and-1-1/

[8] 2015. OpenSSL: OpenSSL 1.0.1 Release Notes. Retrieved April 13, 2020 from https://web.archive.org/web/20150120120428/https://www.openssl.org/news/openssl-1.0.1-notes.html

[9] 2020. *intel/cve-bin-tool*. Intel Corporation. Retrieved April 12, 2020 from https://github.com/intel/cve-bin-tool

[10] 2020. List of free and open-source software packages. *Wikipedia*. Retrieved April 15, 2020 from https://en.wikipedia.org/w/index.php?title=List_of_free_and_open-source_software_packages&oldid=950606447

[11] /news/vulnerabilities-0.9.8.html. *OpenSSL*. Retrieved April 13, 2020 from https://www.openssl.org/news/vulnerabilities-0.9.8.html

[12] SupportedVersions – OpenVPN Community. Retrieved April 13, 2020 from https://community.openvpn.net/openvpn/wiki/SupportedVersions

[13] Hardening – OpenVPN Community. Retrieved April 13, 2020 from https://community.openvpn.net/openvpn/wiki/Hardening

[14] Compatibility Changes in MongoDB 3.0 — MongoDB Manual. *https://github.com/mongodb/docs/blob/v4.2/source/release-notes/3.0-compatibility.txt*. Retrieved April 13, 2020 from https://docs.mongodb.com/manual/release-notes/3.0-compatibility

[15] Support Policy. *MongoDB*. Retrieved April 13, 2020 from https://www.mongodb.com/support-policy

[16] MySQL :: MySQL Workbench Manual :: 5.3.9 Updating Old Authentication Protocol Passwords. Retrieved April 13, 2020 from https://dev.mysql.com/doc/workbench/en/wb-mysql-connections-secure-auth.html
[17] Oracle Lifetime Support Policy. Retrieved April 13, 2020 from http://www.oracle.com/us/support/library/lifetime-support-technology-069183.pdf